H E S B A L

by Jay Balakrishnan

T A B L E O F C O N T E N T S

Human Engineered Software
3748 Inglewood Blvd. Room 11
Los Angeles, California 90066

1.0 Introduction

1.1 Overview

Thank you for purchasing the 6502 Assembler Package, a product of Human Engineered Software. This manual will describe the capabilities, and use of HESBAL for both a PET/CBM and a VIC. It is not intended to be a manual for those who wish to learn 6502 assembler language programming. An excellent book of that type is Lance Levanthal's "6502 Assembly Language Programming", published by Osborne/McGraw-Hill.

HESBAL is supplied as a one-pass assembler. Both one- and two- pass assemblers have to have zero-page variables defined before they are used. Therefore, one might as well define ALL (both one- and two-byte) variables before they are used and then use a one-pass assembler, because it will run much faster than a two-pass assembler. Only data have to be defined first, you can still have any number of forward branches. Using disk I/O makes a two-pass assembler more practical. Should you want to have a two-pass assembler, HESBAL can be made into one, with just a few simple program modifications. They are explained in full in Section 6.0 of the HESBAL Program manual.

The one data file needed by HESBAL will have to be created on your PET/VIC. The instructions for that will be given shortly. Since you will want to use HESEDIT and HESBAL many times, alternating back and forth, we suggest that you take a few minutes to get things set up so that they will always be easy to LOAD. LOAD "HESBAL" and then SAVE it at the beginning of another tape (this way, you can also use the HES-supplied tape as a backup). Now the data file that HESBAL needs should be created. LOAD "HESBAL-DATAGEN" and RUN it. The file that will be created should be written right after the previously saved HESBAL. For possible program modifications to custom-tailor HESBAL-DATAGEN, see section 7.4 in the HESBAL program manual. Now you are ready to run HESEDIT to create some assembler source which can then be fed into HESBAL. For help in using HESEDIT, please refer to it's User Manual.

The following information is just a brief description of what happens when you run HESBAL. For more details, please see Section 3.0. When you are ready to run HESBAL, LOAD "HESBAL" and RUN it. After printing the copyright notice, the program will want to read the "HESDATA" data file which was created by you earlier (creation of HESDATA has to be done only once, unless you modify any of the data in it). Since you wrote HESDATA right after HESBAL, it can read it right away without any positioning of the tape. The HESDATA will be read in (it takes about 90 seconds) and then the number of free bytes available will be displayed. This is the number of bytes which is available for your symbol table.

HESBAL starts out assuming that you will assemble your code into the second cassette buffer, which begins at $033A (a dollar sign "$" before a number means that it is a hexadecimal number) and is 192 bytes long. If this is not where you want the code to be assembled, you will have to set aside the number of bytes you need. The method for this is explained in detail in Section 3.2. Note that $033A is the start of the second cassette buffer on the PET, but is part of the first cassette buffer of the VIC, so you should not begin assembly at the default. Use ORG (section 2.3.2) to start the assembly elsewhere. The total number of

bytes available for the symbol table and your code on an 8K PET is currently about 1000 bytes (about 500 bytes less on an 8K VIC), plus 192 in the second cassette buffer. After printing the number of bytes available for the symbol table, you will be asked how many labels there will be in the source code. If the exact number is not known, give an estimate on the higher side.

Next you will be asked for the name of the HESEDIT data file which contains your assembler source. After that, HESBAL will start reading that file, so replace the cassette containing the HESBAL program with the tape that contains your HESEDIT data file. It is easier if you press the STOP button on the tape recorder after HESBAL finishes reading HESDATA. That way, you will get a prompt to "PRESS PLAY ON TAPE #1", which will give you time to put in the tape containing the HESEDIT file. When you press PLAY, HESBAL will read the assembler source statements and assemble them.

1.2 Notation

In describing the syntax of HESBAL, certain conventions are used. These characters themselves should not be typed in. They are -

```
[ ]   - indicates that the item enclosed is optional
{ }   - indicates that one of the items must be chosen
...   - indicates that the preceding may be repeated
        as often as desired
 /    - indicates that one or the other must be chosen
```

A dollar-sign($) before a number, such as $033A, means that the number is a hexadecimal number.

2.0 Syntax

2.1 Format of the Assembler Line

```
[Label]   Opcode  Operands  [[;] Comments]
```

One or more spaces must separate the label, opcode, operands and comments. If the first non-blank character is a semi-colon (;), the entire line is taken to be a comment. Elsewhere, the semi-colon is optional. That is, the comments come after the operands, if any, and they do not have to be preceded by a ';'. On the VIC, because of the 22 characters per line limitation, you will probably not have enough room to put comments on the same line; put comments on separate lines. Comments may consist of anything.

LABEL

If provided, it may be from 1 to 6 characters in length (this maximum size can be easily changed, as explained later). The first character of the label must be alphabetic and must begin in column 1 of the input. The remaining characters may be alphabetic or numeric. The same label may not be defined more than once. A label may appear on any statement. Because some operands use the letters 'A','X', and 'Y', which are names of the 6502 registers, these letters may not be used as labels. If a label is not given, then you must leave at least one blank at the beginning of the line, and then the opcode may begin in column 2 or after. The following

examples should clarify legal and illegal labels:

```
ADD2IT     - legal
2ADD       - illegal, first character is not alphabetic
X          - illegal, label cannot be A, X, or Y
CALCULATE  - illegal, more than 6 characters long
AXY        - legal
```

OPCODE

This is the standard 3 letter 6502 mnemonic opcode. All 56 opcodes are accepted, as well as 4 pseudo-opcodes.

OPERAND

All 12 of the standard MOS Technology 6502 addressing modes are accepted. Their formats (numbered for future reference, 0 thru 11) are:

No.	Type	Examples
0	Immediate	AND #10
1	Zero-page Direct	AND 61
2	Zero-page Indexed with X	AND $32,X
3	Absolute	AND 7122
4	Absolute Indexed with X	AND 7122,X
5	Absolute Indexed with Y	AND $1FB,Y
6	Indirect, pre-indexed with X	AND (57,X)
7	Indirect, post-indexed with Y	AND ($32),Y
8	Implied	CLC
9	Accumulator	ASL A
10	Relative	BNE -13
11	Indirect	JMP ($6EF)

2.2 Constants

Zero-page constants must have a value from 0 to 255, inclusive. Relative branch constants must be in the range of -126 to +129. All other constants should range in value from 0 to 65535, the maximum value that can be contained in 2 bytes. A negative 0 (-0) will be taken to be positive (+0).

The types of constants allowed currently are decimal and hexadecimal. Hexadecimal numbers are always preceded by the dollar sign ($). If the number is to be signed, the sign should precede the $, i.e. negative 32 hexadecimal would be written as -$32, and not $-32. If the hexadecimal number has an odd number of digits, you do not have to provide a leading zero. Thus, either $033A or $33A is acceptable. In either implementation of HESBAL (one- or two-pass) the assembler still needs to know the number of bytes each instruction and its operands will take. Since some instructions have both the absolute and zero-page addressing modes, if a symbol was used that was undefined at the time, it would not know if the instruction would take 2 or 3 bytes. Therefore it will make the assumption that all undefined symbols are absolute and thus, all zero-page constants must be defined before they are used. Furthermore, if the one-pass version is used, then ALL symbols, both zero-page and

absolute, must be defined before they are used. The only exceptions are labels used in relative branches, JMP or JSR instructions.

The limitation in the one-pass HESBAL that all symbols be defined before they are used is not really the big hinderance it may appear to be at first. This limitation does not say that the LOCATION of all data must come before the program statements, but rather it's DEFINITION to HESBAL. That is, let's say you write a small program, 10 bytes long, starting at $0340. You have a variable that you wish to be at $034A, one byte after the end of the program. It can be done in this manner:

```
          ORG  $034A
SAVEIT    BYT  20
          ORG  $0340
(the whole of the program comes next)
```

This of course does mean that you must know the length of the program or approximate it. If it is small, then there is no problem. At any rate, the idea presented here is to suggest an alternative to putting the definition at the end of the program.

2.3 Pseudo-opcodes

Four pseudo-opcodes are currently implemented -

1. EQU
2. ORG
3. BYT
4. END

Others can be added if their use is frequent enough to warrant the extra bytes needed for their program code.

1. EQU

```
Label  EQU { +/- Integer
           { +/- Symbol
           { *
           { *+/- {Integer}
                  {Symbol }
```

The EQUates pseudo-opcode is used to equate one label to something else. Therefore, the label is mandatory. The "Integer" may be any decimal or hexadecimal number. "Symbol" is any previously defined symbol. The integer or the symbol may be signed or unsigned. The '*' means the current location counter. One important point about EQU - the value assigned to the label will be the value of the constant or the address value if a symbol was given. Some examples:

```
Loc counter (hex)

0300      NINE    BYT  9
0301      FIVE    EQU  5
0301      VALUE1  EQU  *+5
0301      VALUE2  EQU  *+FIVE
```

```
0301      VALUE3  EQU  NINE
0301      VALUE4  EQU  *-$305
```

After these instructions, VALUE1 will have a value of $306, VALUE2 will be $306, and VALUE3, $300. The address of NINE is $300. EQU cannot look at the contents of $300 to find out it's value. Instead, the value of NINE is it's value in the symbol table, it's address - $300. The EQU for VALUE4 will be rejected since it will cause a negative address. However, if the intent is to have VALUE4 be -4, then

```
VALUE4  EQU  -4
```

could be used. In the EQU for VALUE3, if the symbol NINE had not yet been defined, that would have caused an error, because any symbol on the right side of an EQU statement has to have been previously defined.

2. ORG

```
[Label]  ORG { Integer
             { Symbol
             { *
             { * +/- { Integer }
             {       { Symbol  }
```

The ORGinate pseudo-opcode is used to change the value of the assembler's location counter. The operands of ORG are identical to those of EQU's except that negative values are not allowed. When HESBAL starts to RUN, it assumes the value of the location counter to be $033A, which is the beginning of the second cassette buffer. If that needs to be changed, use ORG. ORG is similar to EQU also in that memory is not changed as a result of ORG. One important point about a label on an ORG should be noted. The address value is assigned to the label BEFORE the location counter is set according to the operand. For example:

```
Loc Cntr (hex)
     0400    NEWVAL  ORG  $300
```

NEWVAL will be considered to be at $400, and not $300. A label on an ORG is seldom used (most assemblers prohibit it).

3. BYT

```
[Label]  BYT { +/- Integer } [ , { +/- Integer  ... } ]
             { +/- Symbol  }   { +/- Symbol         }
```

The values given must fit in one byte, i.e. 0 thru 255. As many full values as will fit on one line may be given. If more are needed, additional BYTs can be given on subsequent lines. The address value of the label will be the address of the first byte assembled. No imbedded blanks are allowed in the operands of any statement, except for BYT. So,

```
BYT   20,$0F,    30,         $4F
```

would assemble 20,$0F,30 and $4F. But the comma must always come right after the number, so

```
      BYT  20,   30 ,   $4F
```

will only assemble the "20" and the "30". Since one space came after the "30", the rest is assumed to be comments.

There is no pseudo-op to reserve a certain number of bytes, but that can be easily accomplished by using ORG. Let's say a table of 20 bytes is to be reserved at address $400 and be given the name of TABLE1.

```
  Loc Cntr (hex)

  ????          ORG  $400
  0400  TABLE1  EQU  *
  0400          ORG  *+20
  0414    (continue with the rest of your code)
```

4. END

```
 [Label]  END
```

This pseudo-op is always the very last instruction in the input stream. It signifies that you are finished inputting. The only thing END really does is it goes through the symbol table and checks for symbols that are still undefined. If it finds any, they will be printed out and then you can define them at that time. Let's say you forgot to give the instruction at $400 a label name of ADDIT and you have already made several references to it. END informs you of this undefined symbol, so you provide input from the keyboard (providing input directly from the keyboard in described in detail in section 3.3 a little later) -

```
          ORG  $400
    ADDIT EQU  *
          END
```

Now it is defined, so the END will be accepted and the message "DONE" will be printed. A note about the label on the END statement - it has no effect. It will not be entered into the symbol table and if used, should be for documentation purposes only.

3.0 Operations

3.1 Using HESEDIT Source Files with HESBAL

HESBAL is very convenient to use when the source is a HESEDIT-created data file. When HESBAL asks for it, provide the name of the data file. If you leave out the END pseudo-op from your text, then when the end of your file is reached, HESBAL will ask whether you want to end the assembly or whether you have another data file that contains more assembler source. If you give the name of a second file, then assembly will continue with that file. In this manner, you can assemble as many source files as you want. If you have certain routines that you use often, you can save them separately, and then have them read in at the appropriate times, almost like a macro facility. Note that since HESEDIT can also create general purpose files and not just assembler code, it does not check each line

for valid assembler input. If you have any bad characters in the file, like SHIFTED-SPACE, they will not be caught until you run HESBAL. If your HESEDIT source file is on disk, you can just give the filename when you are prompted for it, the drive number is not necessary. If you want to supply it, then the whole name will have to be in quotes. So if you want to give a filename of FILE1 which is on drive 1, provide: "1:FILE1", INCLUDING the quotes.

3.2 Reserving Memory

As mentioned before, HESBAL initially assumes that you want to start assembling at $033A. This second cassette buffer is 192 bytes long. If your program will fit here, use it by all means. It is well protected from BASIC and it's contents are recoverable in the event of a system crash. If 192 bytes are not enough, then you will have to reserve some memory and protect it so that BASIC doesn't touch it, because HESBAL is written in BASIC. This must be done by you BEFORE HESBAL starts to run, before any strings are used. The above adddress of $033A is OK for a PET but should not be used on VICs.

As you may know, a PET/VIC BASIC program consists of several distinct regions. Besides the BASIC text itself, there are areas for simple variables, array variables, and string variables. The area used to store values of strings is at the very top of memory. This top end of memory is determined once, when the PET is powered-up. This value is then saved in a pointer, at locations 134 ($86) and 135 ($87) in ROM 2.0, 52 ($34) and 53 ($35) in ROMs 3.0 & 4.0 and 55 ($37) and 56 ($38) on the VIC. These values represent the address of the last RAM byte and are stored in the usual way for a 6502 address, first the low-order byte, then the high-order byte. The examples shown are equally valid for OLD ROMs(2.0), NEW ROMs(3.0) and VICs - just different addresses must be used. To find out what the top limit of BASIC currently is, type:

```
PET ROM4.0      PRINT PEEK(52)+PEEK(53)*256
    VIC         PRINT PEEK(55)+PEEK(56)*256
```

Once this value is set at power-up, BASIC will never again change this value, and in fact will believe whatever value is those pointer locations as the top of RAM memory. If we change this value and make it point to a lower address, BASIC will never know the existence of the higher RAM bytes. You can then have your source assemble into this protected area. The following is an example to tie all this together.

Assume you have an 32K PET with 3.0 or 4.0 ROMs. Upon power-up, PET will determine that the highest RAM address is decimal 32768. A PEEK at locations 52 and 53 will show it's contents to be 0 and 128, respectively, which are the low- and high-order bytes for 32768, i.e., 0+128*256. Actually, the highest RAM address is 32767 because you have 32768 bytes, numbered from 0 to 32767. BASIC really stores the highest RAM address plus one in this pointer. If now we need to set aside 500 bytes for our machine language program, subtract 500 from 32767 giving 32267. To determine the high-order byte, divide 32267 by 256 and take the integer part only, 126. The low-order byte is 32267-(126*256) = 11. So, POKE 52,11 and POKE 53,126. Now BASIC thinks that it has only 32267 bytes of RAM and will never touch the 500 bytes above it. After setting the top

of memory limit, issue a "CLR" command in BASIC to make sure all pointers are valid. This is the way you will have to set aside bytes as required by your program. Another example - on an 8K VIC (standard 5K + 3K expansion), location 55 and 56 will show that 7680 is the high end of memory. If your machine language program needs 300 bytes, subtract 300 from 7680, giving 7380, the low byte of which is 212 and the high byte, 28. So POKE 55,212 and POKE 56,28. Don't forget to do the CLR command.

The number of bytes free that is printed out when you first RUN HESBAL represents the number of bytes that are free to be shared between your machine code and the symbol table. The symbol table is a table that is maintained by all assemblers to keep track of the labels and variable names that you use in your assembler code. The symbol table is explained in greater detail in Section 3.0 of the Program Manual, but for now, you should know that each entry in the table takes 10 bytes. Further, for each reference to a symbol that is not yet defined, 2 more bytes, per reference, will be used. If the symbol is already defined, then no matter how many times it is referenced, only 10 bytes will be used. Whenever a label that has been previously referenced, but was undefined until now, is finally defined, the size of that symbol table entry will shrink back to 10 bytes and all those 2 bytes for each reference will be freed up.

To get a rough idea of the number of bytes your machine code program will take up, the following procedure may be used. Count or estimate the number of lines in your program. Assume the worst case, that each instruction takes 3 bytes. Multiply the number of lines in your program by 3, giving the total number of bytes needed by your program. Don't forget to add bytes of data, if any. The amount of memory the symbol table will take up can be calculated as follows. Count the number of labels that you have in the program and multiply by 13 (10 for each entry and 3 bytes for each array descriptor). Take a few seconds to estimate the number of unresolved references there are and multiply it by 2. Add these figures together to arrive at the number of bytes the symbol table will need. When you set aside space at the top of memory, remember to set aside only the amount actually needed by your program itself, as the bytes needed by the symbol table cannot be protected, it must be addressable by BASIC. It is very important to remember to do the setting of BASIC's upper memory limit either before or right after LOADing HESBAL, but before RUNning it.

Whenever a byte is assembled into memory, a check is made to ensure that the POKE did in fact work. This is useful in case you accidentally assemble at an address which is a ROM address (which cannot be modified), or a RAM address that doesn't exist on your machine.

3.3 Correcting Errors

When you have an error in the source code and HESBAL catches it, it will display the appropriate error message and then stop. You can now make the correction right there, if you want, instead of going back into HESEDIT to do it. Let's say a line of your input file contained:

```
LDS  #21
```

Well, since there is no opcode "LDS", HESBAL will indicate that and stop.

You look at it and know that you meant to type in "LDA". Here is what to do - type:

```
Z$=" LDA #21"+CHR$(13):C=LEN(Z$):GOTO 99
```

and you will be able to resume the assembly. Z$ is the line of input that is normally read in from tape or disk. CHR$(13), which is a RETURN, is appended to it. Statement 99 in all HES programs is a way of getting back into the mainstream of the program if it should stop for any reason (including accidently hitting the STOP key).

The above procedure is only good when the error is detected by HESBAL and the program stops. Sometimes, you will want to insert a statement that you left out of the source, or more often, you will want to input an ORG that has been purposefully left out of the input to give you maximum flexibility. In order to input a statement, you only have to hit the SPACE key (or any key except STOP) while HESBAL is running and the NEXT time HESBAL is supposed to read the next line from the file, it will BREAK and stop execution. Now you just give it the statement that you want in Z$ as was just described in the last paragraph and you can resume execution. Your new line will be inserted and then the next source line will come from the file, as usual. Although these features are invaluable when needed, you will probably not use it very much, so in order to conserve memory, these procedures have not been put into HESBAL as a routine. If you find yourself using this run-time input feature a lot, you may want to have a subroutine that queries you for input, appends a RETURN (ASCII 13) and then resumes execution at line 99.

3.4 Saving Machine Language Code

After you have assembled your program, you can save it on tape or disk, so that you can load it in later and execute it. Go into the PET machine language monitor by typing: SYS1024 and use the Save command. For example, to save your code in the second cassette buffer from 826 to 900 to tape:

```
.S "RAZZLEDAZZLE",01,033A,0385
```

Notice that the ending addressing specified was one greater than the real ending address. The monitor requires that.

It might also be convenient to have your machine language program be in DATA statements and have them poked into memory from a BASIC program. Also, if you have a VIC, it does not have a built-in monitor, so you will need to use the next approach. The following program appeared in the October 1981 issue of COMPUTE. When run, it will create DATA statements out of the machine language program.

```
1 INPUT "START ADDRESS";A: INPUT "END ADDRESS";E:Z=2000
2 PRINT "[CLEAR][2 DOWN]"Z"DATA";:IF A>E THEN END
3 FOR A=A TO A+15+(E<A+15)*(A+15-E)
4 PRINT MID$(STR$(PEEK(A)),2)",";:NEXT
5 PRINT "[LEFT] ":PRINT "A="A":E="E":Z="Z+10":GOTO2[HOME]";
6 POKE 623,13:POKE 624,13:POKE 158,2:END
```

For OLD ROMs(2.0), the numbers 623,624 and 158 in line 6 should be replaced by 527,528,525. For the VIC, they should be replaced by 631,632,198. In the above listing, [CLEAR] should be replaced by the CLR key, [2 DOWN] means two cursor down keys, [LEFT] is one cursor left key, and [HOME] means the HOME key.

The above program only produces the DATA statements. You should add the following lines to the program that will be created that contains the DATA statements:

```
10 INPUT "STARTING ADDRESS";S
20 READ D: POKE S,D: S=S+1:GOTO 20
```

The above program will POKE in the DATA statements until there are none left. The poking will end with an "OUT OF DATA" error, which is alright. Once the machine language program has been poked in, you can execute it by: SYS xxxx, where xxxx is the address where the program starts.

4.0 Error Messages

4.1 General Formats

When HESBAL detects an error, if the error detection logic can pinpoint the location of the error in the line, an UP-ARROW in reverse will be printed beneath the character where the error is. For an illegal character, the up-arrow will point exactly at the character. But, if for example, the length of a label is allowed to be only 6 characters in length, and a 10 character label is input, the up-arrow will point right after the 10th character, and not the 7th. Therefore, if an up-arrow is present, the error has occured at or before that point. In some cases, such as an addressing mode that is illegal for that particular instruction, there really is not a particular column at which the error occured, so the up-arrow will not be shown. Much of what has been described will readily become clear after you have run HESBAL a few times.

The error messages, though abbreviated to conserve bytes, should be self-explanatory and be a sufficient aid in determining the cause of the error. But, for further clarification, all errors also have an error number assigned to them so that you can refer to the error list below. An error could occur at three places - in the label, the opcode, or the operand. Also, some general errors occur that are attributable to the whole line. The error messages thus reflect these four types of error areas. The general error numbers have only one digit, 0 thru 9. All others have 2 or 3, the first digit denoting which part of the line the error occured in - 1 = label, 2 = opcode, and 3 = operand. The following is a list of all error messages currently assigned.

4.2 Error List

GENERAL ERRORS

1 - input line was too long (more than 35 characters)
2 - entire line cancelled by user - not an error (only when input is from the keyboard, not HESEDIT)

3 - an illegal character was found, it was not expected there
4 - when END encountered, one or more referenced labels was still undefined

LABEL ERRORS

11 - length of label greater than value allowed (HES supplied value is 6)
12 - one of the following illegal labels used - A,X,Y
13 - a label was required on this statement, but none was given
14 - duplicate label - another label has already been defined with the same name

OPCODE ERRORS

21 - the given opcode or pseudo-opcode is not valid

OPERAND ERRORS

31 - length of symbol greater than value allowed (HES supplied value is 6)
32 - one of the following illegal symbols used - A,X,Y
33 - the opcode required an operand, but none was given
34 - the value given in the operand exceeds 65535, which is the largest value containable in 2 bytes
35 - length of a hex integer exceeds 4 digits (value greater than $FFFF)
36 - undefined symbol found on a non-branch instruction (occurs only in the one-pass version)
37 - illegal use of asterisk (*)
38 - addressing mode of Immediate is illegal for STA
39 - addressing mode of Absolute Indexed with X is illegal for LDX, STY and LDX
310 - addressing mode of Zero-page indexed with X is illegal for STX, and LDX
311 - unknown addressing type or addressing type not allowed for this opcode
312 - illegal operand given for relative branch instruction
313 - operand value for a relative branch is not within the limit (-128 to +129). Also, if a relative branch refers to an undefined label, which is then defined, but more than 129 bytes after the first reference
314 - addressing type other than Absolute given for JSR
315 - Sign (+ or -) not allowed here or is used improperly
316 - operand in pseudo-opcode is given incorrectly
317 - operand value given exceeded one byte value (255)

SEVERE ERRORS

These are errors which cause further execution to be aborted. A message is printed, and execution is stopped.

"POKE ERR AT" - Every byte POKEd is verified to see if indeed the POKE worked. It may not have, if you are assembling at an address which turns out to be a ROM address or an address for which your PET doesn't have RAM there. The bad address is given right after this message.

"TOO MANY REF" - Each entry in the symbol table takes 10 bytes, plus 2 for each unresolved reference. A string can only be 256 characters long, so only 123 ((256-10)/2) unresolved references to a single label is allowed. You get this messages when more than 123 references are made to one undefined label. This should be more than adequate for most programs.

"TABLE FULL" - At the beginning of the assembly process, you will be asked how many labels will be used in the program. This number is used to dimension the symbol table array. In strict fact, if your answer is, for example 10, the table dimensioned 10 will have 11 elements, because the 0th element counts also. Therefore, if space is critical and you have only 10 labels, reply 9 and 10 labels can be entered. When you attempt to input one more label than was declared at the beginning, this message will print out and execution will stop.

H E S B A L

by Jay Balakrishnan

T A B L E O F C O N T E N T S

1.0 Variables

A - address type (index)
A$ - current address type, built through concatenation
A$() - array containing all possible addessing types
AL - low order byte of a 2 byte address
AH - high order byte of a 2 byte address
B - asterisk (*) flag
B$ - constant having value of all spaces, with length equal to D
C - length of Z$, current input line
C$ - current character from input line
D - maximum length of symbols and labels (HES-supplied value = 6)
E - error flag, assigned the value of an error when an error occurs
EF - end of file flag
F - opcode flag
G - sign flag
G$ - general purpose string
H - constant value of 256
H$ - hex convertion subrtn returns hex number of decimal number in H$
HX$- constant value of "0123456789ABCDEF" (all hex digits)
I - general purpose, index
I$ - current integer from operand, if any
J - general purpose, index
K - general purpose, index
L - index into the symbol table, L$()
L$ - current label
L$()- symbol table
L9 - dimension of symbol table, L$()
LS - index into L$() for a label
LV - index into L$() for a symbol (in operand)
M - constant value of -1
M$ - current mnemonic opcode
M1$- 1st character of opcode
MN%()- opcode mask and opcode flag array
MN$- contains list of all valid opcodes
N - general purpose, index
O$()- valid addressing types for 8 categories of opcodes
OC - current opcode value
OM%()- opcode mask table, used to AND with opcode, corresponding to A$
P - current assembly address counter
PL - used as offset into Z$ to print UP-ARROW to show an error
Q - pseudo-opcode indicator
R - value of expression in the operand
S - current state, gotten from the finite state array, S$
S$ - finite state table, kept in string to save memory
ST - PET's status indicator
T - general purpose & 2nd index into S$ to find current character
T$ - label or symbol type
TS$- label or symbol type, saved
TV$- type of symbol name, saved from T$
U - constant value of 1
V - instruction length of current opcode (1,2 or 3 bytes)
V$ - variable name, built through concatenation
VT$- values of T, index of current character, in encoded form
W - next free slot available in the symbol table

X - used to pass parameters to a called subroutine
Y - column counter, used to identify where an error occurs
Z - constant value of 0
Z$ - current input line

2.0 Subroutines

For those subroutines that pass data back and forth between the called and the calling, the INPUT and OUTPUT sections describe these parameters.

1000 thru 1099 - Handle a label

PROCESS. This routine calls 2130 to syntax check a label whenever one is found. If the syntax is OK, the symbol table is searched, by calling 2140. If a label with the same name exists, it will be a duplicate label error. Else it will return normally.

1100 thru 1199 - Handle Opcode

This routine searches MN$, which contains all the valid opcodes, to see if the current opcode is a valid one. If it is not, then a check is made to see if it is one of the pseudo-opcodes. If not, then the opcode is an invalid opcode. If it is valid, the opcode flag, the opcode value, and some other values are set.

1200 thru 1399 - Handle Operand

PROCESS. This routine has to figure out what kinds of operands have been given. First it determines whether the operand is an integer, hex number or a variable, and builds I$, H$, or V$, respectively. The numbers are checked to ensure that they are within the valid range. In the one-pass assembler, an undefined symbol in a non-branch opcode (branch opcodes are all relative branches and JMP and JSR) is not allowed. If the number given is valid, it must be determined if it is a single byte value (max 255) or 2 byte value (max 65535) and accordingly, the address type, A$ is built. A "P" is concatenated to A$ for 1-byte values and a "Q" for 2-byte values.

1500 thru 1899 - Validate operands

PROCESS. This is a long routine. It must validate the given operands against all known operand possibilities and exceptions. The exceptions are mainly with the opcodes STA, LDX, LDY, STX and STY. The operand is validated to make sure that the given opcode allows the kind of operand given. For example, assume that routine 1200 had determined that the addressing type was zero-page indexed with X. If the opcode given was STX, then the operand would be illegal. Checks of pseudo-opcodes are also done.

1900 thru 1999 - Print out undefined symbols at "END"

PROCESS. When an "END" has been found in the input data and assembly is complete, all symbols still undefined will be printed out.

2100 thru 2109 - Initialize variables

PROCESS. This routine is called just before beginning to process each line of input.

2110 thru 2119 - Convert a decimal number to it's hex value

INPUT. X - decimal value to be converted to hex
OUTPUT. H$ - contains value of X, in hex
PROCESS. Simply converts a decimal number to hex.

2120 thru 2129 - Calculate low and high order bytes of a 2-byte value

INPUT. X - 2-byte value
OUTPUT. AL and AH, the low and high order byte values of X, respectively
PROCESS. Separates a two-byte value into it's low and high order bytes.

2130 thru 2139 - Syntax check a label

INPUT. G$ - label name to be checked
OUTPUT. E - error flag is set if there was an error, else left alone
PROCESS. This routine makes sure that a label's length is OK, and that it is not one of the prohibited labels (A,X,Y)

2140 thru 2149 - Search the symbol table for a label

INPUT. G$ - the label to be searched
OUTPUT. L - set to index into table, if found
T$ - set to the label type
PROCESS. This routine does a simple, linear search for label G$, in the symbol table, L$(). If found, L will be the index into L$() and T$ will be the label type.

2150 thru 2159 - Add label into symbol table

INPUT. G$ - the label to be entered into the symbol table
OUTPUT. none
PROCESS. This routine is used to enter a new label into the symbol table, L$(). It makes sure that the table is not full and then adds it.

2160 thru 2169 - concatenate "P" or "Q" to A$

INPUT. R - the value of the operand
OUTPUT. A$ - addressing type
PROCESS. This routine determines whether the value given in the operand is a single byte or a two byte operand. A character is appended to A$, the addressing type, depending upon this test. "P" means it is a 1-byte value, or in other words, a zero-page addressing mode. "Q" means 2-bytes or absolute addressing mode. If the opcode is "JSR" or "JMP", then regardless of the value, it will be "Q", since they have no zero-page addressing modes.

2170 thru 2179 - Add an unresolved reference to a label

INPUT. AL and AH - low and high values of the address

OUTPUT. none
PROCESS. Whenever a reference is made to an unresolved label in the one-pass version, a note of that must be made. This is done by appending the address of this unresolved reference to the end of the entry in the symbol table for that label. The base (fixed) part of the label entry is always 10 bytes. Therefore, since the maximum length of a string is 256, 246 bytes are available for this. Each unresolved reference takes up two bytes, for the address. So, only 123 (246/2) unresolved references to one single label may be made. This routine ensures that there is enough room in the label entry to add this unresolved reference and then proceeds. Note that all this is done only on UNRESOLVED references. For references to previously defined labels, the label entry will always stay at 10 bytes, no matter how many references are made to it.

2180 thru 2189 - POKE a byte and verify it

INPUT. X - address where the byte is to be POKEd
T - value that is to be POKEd at X
OUTPUT. none
PROCESS. This routine is called whenever the assembler has to POKE a value into memory. After the POKE, a PEEK is done to verify that the POKE worked. It might not, if for example, you were assembling into an address that was the address of a ROM area, or if it was a RAM area, but there were no RAM there in your PET. If the verification test is not passed, execution must stop.

2190 thru 2199 - Handle signed numbers

INPUT. R - absolute value of operand
G - sign flag
OUTPUT. none
PROCESS. This routine checks to see if a sign was given in the operand. The value of G will be 0 if none given, -1 if "-" sign was given and +1 if a "+" was given. The sign on numbers are allowed only for one byte values. A two's-complement number is created, as required by the 6502. Two's-complement means the high order bit is used for the sign bit - 0 means a positive number and 1 means a negative number. The remaining 7 bits hold the value . Because of this arrangement, the range for a signed, one byte number is -128 to +127.

2300 thru 2399 - Get input

INPUT. Input comes from a tape/disk file
OUTPUT. Z$ - contains the current input line
PROCESS. This routine reads in input from tape or disk and assigns the line to Z$. This routine will also STOP before reading the next line of input, if any key has been hit before the program gets to this routine. This is so that you can stop the program when you want to, and provide input directly, from the keyboard. It is explained in detail in the User Manual, in Section 3.3. This entire subroutine can also be replaced by one that does not accept the input from tape/disk, but rather from the keyboard. See section 7.3 in the Program Manual for that.

2500 thru 2699 - One-time processing initializations

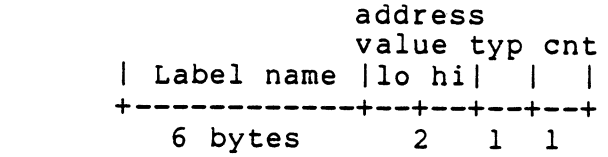
PROCESS. This routine is called only once, at the start of the program. It reads in the data file, HESDATA, which contains all the tables, finite state array, etc and initializes those arrays and variables in the program. It also asks for the dimension of the symbol table.

2700 thru 2799 - Get the name of the input data file

PROCESS. This routine merely asks for the names of the input file and opens the file.

3.0 Symbol Table

The symbol table in the program is kept in the one-dimensional array variable, L$(). It is dimensioned to a size specified by you at the beginning of the program. Thus, when the question, "NUMBER OF LABELS?" is asked, if you reply 10, the table will be dimensioned L$(10). Note that this actually allows 11 labels, since the 0th element is used, so adjust your answer accordingly, if desired.



The typical symbol table entry format is shown in the figure above. Each entry takes up 10 bytes (throughout this manual, the maximum length for a label will be assumed to be 6 bytes long, which is the value as supplied by HES. This value is very easily changed, as described later on in Section 7.4). The "label name" is any symbol name that appears in the label field (columns 1-6) or in the operand. The "address value" is a pointer to the location in memory that is associated with the label. Note that the two-byte address value is stored with the low-order byte first, then the high-order byte, which is the standard for 6502 addresses. In the case of a label that is not yet defined, this address value will be zero.

The "type" is the type of symbol this is. It can have one of three values:

L - a regular label or symbol
E - a label that was defined on an EQUates statement
U - an undefined label

The "cnt" is a count of the number of references to this label when it is unresolved. When the first reference is made to an unresolved label, the label is added to the table with it's name, address value of zero, type of "U", and a count of 1. Also, added on to the end of the 10 byte root entry will be an additional 2-byte address, which is the address of the instruction that is referencing the undefined label. For example,

```
Loc cntr (hex)
  033A        BCC  NEWLAB
```

Assuming that NEWLAB is at this point undefined, the count would be 1 and the address value would be zero. The actual entry would look like this:

```
|NEWLAB|00|00|U |01|3A|03|
+------+--+--+--+--+--+--+
```

When it is finally defined, the "U" will become either an "L" or an "E", the count will be 0 and the bytes after the 10th will all be dropped.

4.0 Addressing Modes

The 6502 has 12 addressing modes. They are listed below, along with the number of bytes that each takes. In addition, several of the values used by HESBAL are shown.

	Addressing Type	# of bytes	mask for bits 2,3,4	8 bit mask	8 bit mask(dec)
0	Immediate	2	010	0000 1000	8
1	Direct	2	001	0000 0100	4
2	Zero page indexed with X	2	101	0001 0100	20
3	Extended direct	3	011	0000 1100	12
4	Absolute indexed with X	3	111	0001 1100	28
5	Absolute indexed with Y	3	110	0001 1000	24
6	Pre-indexed with X, indirect	2	000	0000 0000	0
7	Post indexed with Y, indirect	2	100	0001 0000	16
8	Implied	1	000	0000 0000	0
9	Accumulator	1	010	0000 1000	8
10	Relative	2	000	0000 0000	0
11	Indirect	3	---	0110 1100	108

Figure 1

The masks will be explained next.

The following symbols are used in the object codes in Table 3-6.

Address-mode Selection:

aaa

000	pre-indexed indirect - (addr.X)
001	direct - addr
010	immediate - data
011	extended direct - addr16
100	post-indexed indirect - (addr).Y
101	base page indexed - addr.X
110	absolute indexed - addr16.Y
111	absolute indexed - addr16.X

bb

00	direct - addr
01	extended direct - addr16
10	base page indexed - addr.X
11	absolute indexed - addr16.X

bbb

001	direct - addr
010	accumulator - A
011	extended direct - addr16
101	base page indexed - addr.X; addr.Y in STX
111	absolute indexed - addr16.X; addr16.Y in STX

cc

00	immediate - data
01	direct - addr
11	extended direct - addr16

ddd

000	immediate - data
001	direct - addr
011	extended direct - addr16
101	base page indexed - addr.Y in LDX; addr.X in LDY
111	absolute indexed - addr16.Y in LDX; addr16.X in LDY

pp — the second byte of a two- or three-byte instruction

qq — the third byte of a three-byte instruction

x — one bit choosing the address mode:

0	direct - addr
1	extended direct - addr16

y — one bit choosing the JMP address mode:

0	extended direct - label
1	indirect - (label)

Figure 2

Reprinted from Lance Levanthal's
"6502 ASSEMBLY LANGUAGE PROGRAMMING"
by permission of
OSBORNE/MC-GRAW-HILL
630 Bancroft Way
Berkeley, California 94710

5.0 Opcode Classes

All of the 6502 opcodes can be put into ten classifications. Seven of them are based upon the opcode classification described on page 3-33 of Lance Leventhal's "6502 ASSEMBLY LANGUAGE PROGRAMMING", which is reprinted in Figure 2. The seven classes of opcodes are:

aaa, bb, bbb, cc, ddd, x and y.

But three more are needed to include all opcodes. The eighth one is for JSR, the nineth for all relative branches and the tenth, for implied modes. These ten classes are tabulated in Figure 3.

	opcode class	bit structure 76543210	# of opcodes in this class	addressing modes in this class
1	aaa	...aaa01	8	0,1,2,3,4,5,6,7
2	bb	...bb1..	4	1,2,3,4
3	bbb	...bbb10	4	1,2,3,4,9
4	cc	...0cc00	2	0,1,3
5	ddd	101ddd..	2	0,1,2,3,4
6	x	0010x100	1	1,3
7	y	01y01100	1	3,11
8	JSR	--------	1	3
9	relative	--------	8	10
10	implied	--------	25	8

Figure 3

Note that for classes 1 thru 7, bits 2,3, and 4 either contain letters or have a constant number in them. The letters represent different bit settings, depending upon the addressing modes. The periods represent different bit settings, depending upon individual opcodes. Thus we can start with an opcode's basic bit structure mask and by changing bits 2,3, and 4, we can have the opcode byte be different for different addressing modes. These different masks were shown in Figure 1, under the heading of "masks for bits 2,3,4".

6.0 Two-Pass Assembler

Although the one-pass assembler is faster and easier to use than a two-pass assembler, occasionally you may want to use a two-pass assembler. HESBAL can be modified to run as both a one- and two-pass assembler if you can afford the memory needed to make the simple program modifications. Note that even in the two-pass version, you must still define all zero-page variables before they are used, since by using the MOS 6502 mnemonics, which are standard, there is no way to tell if an undefined variable is a one or two byte variable. Also, you cannot use the routine described later in Section 7.3 to have input come solely from the keyboard. Since the input will have to be read in a second time, you will have to use HESEDIT to create assembler source that is going to be run through the two-pass version of HESBAL.

The modifications to HESBAL to make it a one- and two-pass assembler will be described shortly. First, an explanation of how to use it. After

RUNing it, you will be asked whether this is a one or two-pass assembly. Reply "2". The rest of the run is very similar to the one-pass. An example will fully illustrate how to assemble in two passes.

Say you have a large program that you want to assemble. You have broken it up into two modules, MODA and MODB, so that you can edit it in HESEDIT on your 8K PET/VIC. When HESBAL asks you "FILENAME or 'END'", reply "MODA". MODA will be read and when end-of-file is reached, reply "MODB" to the question of whether you have more files to be assembled. MODB will then be read in, and on it's end-of-file, this time reply "END", because you have finished with the last module you want assembled. HESBAL will then go into its second pass mode and will want to re-read the source code again, so when it asks you for a filename again, reply "MODA". MODA will be re-read and when done, you should reply "MODB". At the end of MODB, reply "END". Then the second pass will have been completed and your code will have been assembled. If you use the two-pass version often and you can afford the memory, the task can be simplified, if, during the first pass, you have the program keep track of the filenames, and then use these during the second pass, so that you do not have to provide the same names all over again.

Now here are the program modifications to make HESBAL run as one- or two-pass assembler. Unless otherwise mentioned, each line is to be typed in, in its entirety. The spaces in the lines are only for neatness, you should leave them out to save memory.

```
380  IF Q = 4 THEN IF PC = PS THEN PRINT "DONE":STOP
385  IF Q = 4 THEN PC = 2: PRINT "PASS 2": P=826: GOTO 100
485  IF PS=2 THEN IF((N AND 15)<>Z)OR((N AND 16)<>16)THEN N=32:GOTO 480
1030 GOSUB 2140: IF T$ = "L" OR T$ = "E" THEN IF PC = U
     THEN PRINT "DUPLICATE LABEL";:E = 14:RETURN
1520 Y = Z: IF V$ = "" OR M1$ = "J" OR (PS = 2 AND PC = U AND Q = Z)
     GOTO 1540
make the current 1910 into 1920
make the current 1900 into 1910
1900 IF PC = U THEN RETURN
make the current 2150 into 2152
2150 IF PC = 2 THEN RETURN
2605 INPUT "ONE OR TWO PASS ASSEMBLER(1/2)";PS: PC = U
```

The two new variables are PS and PC:
PS - 1 or 2 according to whether this is a one- or two-pass run
PC - the current pass number (1st or 2nd)

7.0 Customizing the Program

7.1 Disk Interfacing

If you bought the program on a diskette, the version on it is the disk version. If you bought the cassette and you want to be able to run it on a disk, you will need to change HESBAL:

```
2710 OPEN 2,8,2,G$+",S,R":EF=Z:RETURN
2540 DIM MN%(55), A$(11), O$(8), OM%(11): OPEN 2,8,2,"HESDATA,S,R"
```

You will also need to change HESBAL-DATAGEN:

```
30 OPEN 2,8,2,"0:HESDATA,S,W"
```

7.2 Printer Interfacing

In order to have a printer print out each line as it is assembled, as well as print out the current address counter, the opcode and the operands in hex, the following program changes can be made to the standard HESBAL. If the two-pass version is used, the changes will cause printing only on the second pass.

```
225  IF C$ = ";" AND M$ = "" THEN GOSUB 3000:GOTO 120
530  IF Q THEN GOSUB 3000:ON Q GOTO 120,640,620
590  GOSUB 3000: X = P:T = OC: GOSUB 2180: X = R: GOSUB 2120:
     ON V GOTO 620,610,600
2510 OPEN 4,4
2606 INPUT "PRINT OUTPUT(Y/N)";Q$
3000 IF PC <> PS OR Q$ <> "Y" THEN RETURN
3005 X = P:GOSUB 2110: PRINT#4, RIGHT$("000"+H$,4);"  ";
3010 IF S = 88 OR Q THEN PRINT#4, SPC(9);:GOTO 3070
3020 X = OC:GOSUB 2110:GOSUB  3100
3030 IF V = U THEN PRINT#4, SPC(6);:GOTO 3070
3040 X = R:GOSUB 2120:X = AL:GOSUB 2110:GOSUB 3100
3050 IF V = 2 THEN PRINT#4, SPC(3);:GOTO 3070
3060 X = AH:GOSUB 2110:GOSUB 3100
3070 PRINT#4," ";Z$:RETURN
3100 PRINT#4, RIGHT$("0"+H$,2);" ";:RETURN
```

7.3 Direct Input from keyboard

When you want to assemble small programs, it may seem like a nuisance to have to run HESEDIT every time you want to make one change. Therefore, there are times it would be convenient if the entire input could come from the keyboard. Program modifications to replace the entire input subroutine will be given next. You may want to create a version of HESBAL that has this feature and SAVE it.

The subroutine that will be given is a generalized GET routine. It gets one character at a time and will allow the use of the DELETE key so that you can make simple changes as you type in text. No other cursor key is allowed. If you have typed too much on one line, and you want to correct something that occurred at the beginning of the line, it may be easier just to have the whole line thrown away and just re-type it. That can be done by holding SHIFT and then RETURN. Normally, you will type in one line of code, hit RETURN and that line will get assembled immediately. As mentioned earlier, this can only work on the one-pass version of HESBAL. As in HESEDIT, 35 characters (17 for VIC) is the longest line that will be accepted. You will still get error messages just like before. Another advantage in using this direct get routine is that any characters that are unacceptable to HESBAL (such as graphics characters) will automatically be ignored in the input routine. Here now is the input subroutine. You should delete all line between 2300 and 2399 and replace those lines with the following:

```
2300 C = Z:Z$ = "": PRINT H$;TAB(5);: PL = 4:H$ = ""
2305 PRINT CHR$(164)CHR$(157);
2310 GET#1, C$: IF C$ = "" GOTO 2310
2320 K = ASC(C$): IF K = 13  THEN PRINT " ": GOTO 2370
2330 IF K = 34 THEN PRINT CHR$(K)CHR$(20);:GOTO 2370
2340 IF K = 20 THEN IF C >= U THEN C = C-U: Z$ = LEFT$(Z$,C):
     PRINT C$;:GOTO 2305
2345 IF K = 20 GOTO 2305
2350 IF K = 141 THEN PRINT " !":E=2:RETURN
2360 IF K < 32 OR K > 95 GOTO  2305
2370 PRINT C$;:C = C + U: Z$ = Z$ + C$
2380 IF C > 35 THEN PRINT: PRINT "TOO LONG";: Y = U:E = U:RETURN
2390 IF K = 13 THEN RETURN
2395 GOTO 2305
2700 OPEN 1,0,1
2710 RETURN
```

The 35 on line 2380 should be changed to 17 for VICs.

7.4 Other Program modifications

1. To change the maximum allowed length of labels, change variable D in line 2500. Also in 2520, make sure that variable B$ contains as many blanks as the new value of D.

2. Modifying HESBAL-DATAGEN

HESBAL-DATAGEN is the program that creates the data file, "HESDATA", that is used in each run of HESBAL. If you have more than 8K, you might want to put the data statements in HESBAL-DATAGEN into HESBAL. This will make it run faster and you won't have to read the data in from tape.

HESBAL searches for each opcode in a sequential manner. The small amount of time saved by using binary or other search techniques does not seem to justify the extra amount of memory required, at least for an 8K system. The order in which the opcodes are positioned with the opcode table was chosen after some study into the frequency of usage of opcodes. They appear in lines 2000 thru 2999 of HESBAL-DATAGEN in declining frequency of usage. However, you may use some opcodes more than others, based upon your own style of coding. Therefore, you may want to change the order in which the opcodes are searched, to minimize search time. Put the opcodes that you use most often at the beginning of the list. Each line contains the mnemonic, it's masked value and it's class. When you move them around, be sure to keep these three values together.

8.0 References and Acknowledgements

1. The SEPT/79 issue of Creative Computing had an article and program code for a 6502 disassembler. The article was written by Anthony Scarpelli, on pages 124-129. Another disassembler for the PET written by Michael McCann is in pages 5:25 to 5:27 of the June/July of 78 issue of MICRO. However, this version uses 6502 mnemonics that are not the standard MOS 6502 mnemonics, so it is not as useful. The best disassembler I have seen is part of EXTRAMON, by Bill Seiler. HES includes, as a courtesy item, a copy of MICROMON (only for the PET

assembler package), which is an extended EXTRAMON, that will run on ROM3.0 or 4.0. The Jan/81 issue of MICRO contains a symbolic disassembler in BASIC, written by Werner Kolbe, on page 23-26.

2. When you are developing assembler code and testing it, you may crash the system many times. Using the ON/OFF switch on the PET often is a strain on the system. So you may want to make a RESET switch for it, so that you can reset the system, without clearing the memory and wiping out your program. The SEPT/80 issue of KILOBAUD contains an article on pages 36-37 by James Strasma on making one. Also, in the Fall 79 issue of COMPUTE, which is issue 1, there is an article by Jim Butterfield on software reset, on page 89. It is only for "NEW" ROMs, 3.0.

3. The excellent routines for converting a decimal number to hex in lines 2110 thru 2115 and for converting a hex number to decimal in line 1350 are from Gregory Yob's PET column in Creative Computing, in the JAN/80 issue, on page 148.

4. I would like to express my appreciation to Tom Rugg for his assistance and encouragement and to David Malmberg for his helpfulness. Also, I thank Georgette Psaris, of Osborne/McGraw-Hill for granting permission to reproduce page 3-33 of Lance Leventhal's "6502 ASSEMBLY LANGUAGE PROGRAMMING".

5. A simple monitor for the VIC appeared in the Jan. 82 issue of COMPUTE on page 176. It should be a valuable tool for the 6502 assembler programmer.

9.1

```
0     PRINT "[CS][CD]HESBAL 8K TAPE R1.2 3/1/81
1     PRINT "(C)1980 HUMAN ENGINEERED SOFTWARE
2     PRINT ,"BY JAY BALAKRISHNAN
9     GOSUB 2500:
      GOTO 100
99    GOSUB 2100:
      GOTO 140
100   IF E
      THEN PRINT " ERROR,#"E:
        IF Y
        THEN PRINT "[CU][CU]"TAB(PL+Y)"[RO]^[CD][CD]
110   IF E
      THEN STOP
120   X=P:
      GOSUB 2110:
      GOSUB 2100:
      GOSUB 2300:
      ON EGOTO 100,120
140   IF C=U
      GOTO 120
150   C=C-U:
      FOR N=U TO C:
        Y=N:
        C$=MID$(Z$,N,U):
        IF F=Z
        THEN S=88:
          GOTO 210
155     IF C$<" " OR C$>"←"
        THEN T=Z:
          GOTO 170
160     T=ASC(MID$(VT$,ASC(C$)-31,U))
165     IF T=Z
        THEN IF C$=";"
          THEN IF M$=""
            THEN S=88:
              GOTO 210
170     IF T=Z
        THEN S=Z:
          E=3:
          GOTO 190
180     S=ASC(MID$(S$,(S-U)*13+T,U)):
        IF S=88
        GOTO 210
190     IF S=Z
        THEN PRINT "BAD CHAR";:
          E=3:
          GOTO 210
200     ON SGOSUB 9,240,250,260,270,280,290,300,310,315,330,310,320,320,3
        10,310,340
210     IF E OR S=88
        THEN N=C
220   NEXT N:
      IF E
      GOTO 100
225   IF C$=";" AND M$=""
      GOTO 120
230   GOTO 350
240   L$=L$+C$:
```

```
      RETURN

250   GOSUB 1000:
      RETURN

260   M$=M$+C$:
      RETURN

270   GOSUB 1100:
      RETURN

280   V$=V$+C$:
      RETURN

290   I$=I$+C$:
      RETURN

300   H$=H$+C$:
      RETURN

310   A$=A$+C$:
      RETURN

315   RETURN

320   GOSUB 1200:
      GOSUB 1850:
      RETURN

330   G=44-ASC(C$):
      RETURN

340   B=U:
      RETURN

350   GOSUB 1100:
      IF E
      GOTO 100
360   GOSUB 1200:
      IF E
      GOTO 100
370   GOSUB 1500:
      IF E
      GOTO 100
380   IF Q=4
      THEN PRINT "DONE":
        STOP
390   IF Q=Z
      THEN V=VAL(MID$("222333221123",A+U,U))
410   IF L$=""
      THEN IF Q=U
        THEN PRINT "LABEL";:
          E=13:
          GOTO 100
420   IF L$=""
      GOTO 530
440   X=P:
      IF Q=U
      THEN X=R
```

```
450  GOSUB 2120:
     G$=L$:
     L=LS:
     T$=TS$:
     IF T$<>"U"
     THEN GOSUB 2150:
       GOTO 530
460  J=ASC(MID$(L$(L),D+4,U)):
     FOR K=U TO J
470    I=ASC(MID$(L$(L),D+4+K+K,U))*H+ASC(MID$(L$(L),D+4+K+K-U,U)):
       N=PEEK(I)
480    IF N=32 OR N=76 OR N=108
       THEN X=I+U:
         T=AL:
         GOSUB 2180:
         X=I+2:
         T=AH:
         GOSUB 2180:
         GOTO 510
490    N=P-(I+2)
492    IF N>129
       THEN PRINT "REF TOO FAR AT ";:
         X=I:
         GOSUB 2110:
         PRINT H$;:
         E=313:
         K=J:
         GOTO 510
495    IF N<Z
       THEN N=H+N
500    X=I+U:
       T=N:
       GOSUB 2180
510  NEXT K:
     IF E
     GOTO 100
515  T$="L":
     IF Q=U
     THEN T$="E"
520  L$(L)=LEFT$(L$(L),D)+CHR$(AL)+CHR$(AH)+T$+CHR$(Z)
530  ON QGOTO 120,640,620
540  IF V$=""
     GOTO 570
545  IF V$=L$
     THEN TV$="L":
       R=P
550  L=LV:
     T$=TV$:
     IF T$="X"
     THEN T$="U":
       G$=V$:
       AH=Z:
       AL=Z:
       GOSUB 2150
560  IF T$="U"
     THEN X=P:
       GOSUB 2120:
       GOSUB 2170
570  IF M$="JSR" OR (A=Z AND (F=4 OR F=5))
```

```
      GOTO 590
580   OC=OC OR OM%(A)
590   X=P:
      T=OC:
      GOSUB 2180:
      X=R:
      GOSUB 2120:
      ON VGOTO 620,610,600
600   X=P+2:
      T=AH:
      GOSUB 2180
610   X=P+U:
      T=AL:
      GOSUB 2180
620   P=P+V
630   GOTO 120
640   P=R:
      GOTO 120
1000  IF L$=""
      THEN RETURN

1010  G$=L$:
      GOSUB 2130:
      Y=U:
      IF E
      THEN RETURN

1020  L$=G$:
      T$="X":
      IF W=M
      GOTO 1040
1030  GOSUB 2140:
      IF T$="L" OR T$="E"
      THEN PRINT "DUPLICATE LABEL";:
        E=14:
        RETURN

1040  LS=L:
      TS$=T$:
      RETURN

1100  IF F<>M
      THEN RETURN

1110  T=3:
      E=21:
      J=55:
      FOR K=Z TO J:
        IF M$=MID$(MN$,K*T+U,T)
        THEN X=K:
          E=Z:
          K=J
1120  NEXT :
      IF E
      THEN GOSUB 1160:
        IF E
        THEN PRINT "OPCODE";:
          RETURN
```

```
1130 M1$=LEFT$(M$,U):
     IF TS$="X"
     THEN TS$="L"
1140 IF Q
     THEN F=10:
       RETURN

1150 F=INT(MN%(X)/1000):
     OC=MN%(X)-F*1000:
     RETURN

1160 IF M$="EQU"
     THEN Q=1
1170 IF M$="ORG"
     THEN Q=2
1175 IF M$="BYT"
     THEN Q=3
1180 IF M$="END"
     THEN Q=4
1190 IF Q
     THEN E=Z
1195 RETURN

1200 IF R<>M
     THEN RETURN

1210 IF F=Z OR Q=4
     THEN RETURN

1220 Y=Z:
     IF I$>""
     GOTO 1270
1230 IF V$>""
     GOTO 1290
1240 IF H$>""
     GOTO 1340
1250 IF B AND G=Z
     THEN R=Z:
       RETURN

1260 PRINT "OPERAND";:
     E=33:
     RETURN

1270 R=VAL(I$):
     IF R>65535
     THEN PRINT "NUMBER";:
       E=34:
       RETURN

1280 GOSUB 2160:
     RETURN

1290 IF V$="A"
     THEN A$=A$+V$:
       V$="":
       RETURN

1310 Y=N:
```

```
      G$=V$:
      GOSUB 2130:
      IF E
      THEN E=E+20:
        RETURN

1320  V$=G$:
      T$="X":
      GOSUB 2140:
      TV$=T$:
      LV=L
1325  IF T$="L" OR T$="E"
      THEN R=FN R(Z):
        GOSUB 2160:
        RETURN

1330  R=Z:
      A$=A$+"Q":
      RETURN

1340  J=LEN(H$):
      IF J>4
      THEN PRINT "HEX NUMBER";:
        E=35:
        RETURN

1350  R=Z:
      T=16:
      FOR I=U TO J:
        FOR K=U TO T:
          IF MID$(H$,I,U)=MID$(HX$,K,U)
          THEN R=R*T+K-U:
            K=T
1360  NEXT K,I:
      GOSUB 2160:
      RETURN

1500  IF F=Z
      THEN A=8:
        RETURN

1510  IF F=9
      GOTO 1700
1520  Y=Z:
      IF V$="" OR M1$="J"
      GOTO 1540
1530  IF TV$<>"L" AND TV$<>"E"
      THEN PRINT "UNDEFINED NAME";:
        E=36:
        RETURN

1540  ON QGOSUB 1800,1800,1850,1900:
      IF Q
      GOTO 1650
1550  IF G AND A$<>"#P"
      THEN E=311:
        GOTO 1650
1555  GOSUB 2190
1560  IF A$="#P"
```

```
      THEN IF M$="STA"
        THEN E=38:
          GOTO 1650
1570  IF A$="Q,X"
      THEN IF M$="LDX" OR M$="STY" OR M$="STX"
        THEN E=39:
          GOTO 1650
1580  IF A$="P,X"
      THEN IF M$="STX" OR M$="LDX"
        THEN E=310:
          GOTO 1650
1590  A=M:
      T=11:
      FOR K=Z TO T:
        IF A$=A$(K)
        THEN A=K:
          K=T
1600  NEXT :
      IF A=M
      THEN IF A$="P,Y"
        THEN IF M$="LDX" OR M$="STX"
          THEN A=2:
            GOTO 1660
1605  IF A=M
      THEN E=311:
        GOTO 1650
1610  IF A$="Q,Y"
      THEN IF M$="LDX"
        THEN A=4
1620  G$=RIGHT$(STR$(A),U):
      IF A=11
      THEN G$="B"
1630  E=311:
      J=LEN(O$(F)):
      FOR K=U TO J:
        IF G$=MID$(O$(F),K,U)
        THEN E=Z:
          K=J
1640  NEXT
1650  IF E
      THEN PRINT "OPERAND";:
        RETURN

1660  IF B
      THEN IF Q=Z
        THEN E=37:
          GOTO 1650
1670  RETURN

1700  IF A$<>"P" AND A$<>"Q"
      THEN E=312:
        GOTO 1650
1710  IF B
      THEN IF TV$="X"
        GOTO 1530
1720  IF TV$="L" OR TV$="E"
      THEN R=R-(P+2):
        IF R<Z
        THEN R=ABS(R):
```

```
          G=M
1730 IF R>129+(G=M)*3
     THEN E=313:
       GOTO 1650
1740 GOSUB 2190:
     A=10:
     GOTO 1670
1800 IF G=M
     THEN IF B=Z AND Q=2
       THEN E=315:
         RETURN

1810 IF A$<>"" OR B=Z
     THEN IF A$<>"Q"
       THEN E=316:
         RETURN

1820 IF G=M
     THEN IF B
       THEN R=-R:
         GOTO 1840
1830 GOSUB 2190
1840 R=R+P*B:
     IF R<Z OR R>65535
     THEN E=34:
       RETURN

1845 RETURN

1850 IF Q<>3
     THEN A$=A$+C$:
       RETURN

1860 IF R>=H
     THEN E=317:
       GOTO 1890
1870 Y=N:
     IF A$<>"P" OR S=14 OR B
     THEN E=316:
       GOTO 1890
1880 GOSUB 2190:
     X=P+V:
     T=R:
     GOSUB 2180
1885 S=5:
     V=V+U:
     A$="":
     I$="":
     H$="":
     V$="":
     R=M:
     G=Z:
     RETURN

1890 PRINT "OPERAND";:
     RETURN

1900 FOR K=Z TO W:
       IF MID$(L$(K),D+3,U)="U"
```

```
        THEN PRINT "UNDEF-"LEFT$(L$(K),D):
          L=4
1910 NEXT :
     RETURN

2100 S=U:
     E=Z:
     F=M:
     R=M:
     G=Z:
     V=Z:
     Q=Z:
     B=Z
2105 L$="":
     M$="":
     TS$="":
     TV$="":
     A$="":
     V$="":
     I$="":
     RETURN

2110 T=16:
     H$=""
2112 K=INT(X/T):
     J=X-T*K:
     H$=MID$(HX$,J+U,U)+H$:
     IF K>Z
     THEN X=K:
       GOTO 2112
2115 RETURN

2120 AH=INT(X/H):
     AL=X-AH*H:
     RETURN

2130 J=LEN(G$):
     IF J>D
     THEN E=11:
       GOTO 2136
2132 IF G$="A" OR G$="X" OR G$="Y"
     THEN E=12:
       GOTO 2136
2134 G$=G$+MID$(B$,U,D-J):
     RETURN

2136 PRINT "NAME";:
     RETURN

2140 FOR K=Z TO W:
       IF G$=LEFT$(L$(K),D)
       THEN L=K:
         T$=MID$(L$(K),D+3,U):
         K=W
2145 NEXT :
     RETURN

2150 W=W+U:
     IF W>L9
```

```
      THEN PRINT "TABLE FULL":
        STOP
2153  IF Q=U
      THEN T$="E"
2155  L$(W)=G$+CHR$(AL)+CHR$(AH)+T$+CHR$(Z):
      L=W:
      RETURN

2160  G$=CHR$(80-(R>=H)):
      IF M1$="J" OR Q=U OR Q=2
      THEN G$="Q"
2165  A$=A$+G$:
      RETURN

2170  J=ASC(MID$(L$(L),D+4,U)):
      IF D+4+J+J>=254
      THEN PRINT "TOO MANY REF":
        STOP
2173  K=LEN(L$(L)):
      G$=LEFT$(L$(L),D+3)+CHR$(J+U)
2174  L$(L)=G$+MID$(L$(L),K-(J+J)+U,J+J)+CHR$(AL)+CHR$(AH)
2176  RETURN

2180  POKE X,T:
      IF PEEK(X)<>T
      THEN PRINT "POKE ERR AT ";:
        GOSUB 2110:
        PRINT H$;:
        STOP
2185  RETURN

2190  IF G<>M
      THEN RETURN

2193  IF R>128
      THEN E=314:
        Y=Z:
        RETURN

2196  IF R=Z
      THEN G=U:
        RETURN

2198  R=H-R:
      RETURN

2300  GET C$:
      IF C$<>""
      THEN STOP
2310  IF EF
      THEN GOSUB 2700:
        IF G$="END"
        THEN Z$=" END":
          GOTO 2340
2320  INPUT#2,Z$:
      IF ST=64
      THEN CLOSE 2:
        EF=U
2340  Z$=Z$+CHR$(13):
```

```
      C=LEN(Z$):
      PRINT H$TAB(5)Z$:
      H$="":
      PL=4:
      RETURN

 2500 Z=0:
      U=1:
      W=-1:
      M=-1:
      H=256:
      P=826:
      D=6
 2520 B$="      ":
      HX$="0123456789ABCDEF"
 2530 DEF FN R(X)=H*ASC(MID$(L$(L),D+2,U))+ASC(MID$(L$(L),D+U,U))
 2540 DIM MN%(55),A$(11),O$(8),OM%(11):
      OPEN 2,1,0,"HESDATA"
 2550 FOR K=U TO 221:
        INPUT#2,S:
        S$=S$+CHR$(S):
      NEXT
 2560 FOR K=Z TO 55:
        INPUT#2,MN%(K),G$:
        MN$=MN$+G$:
      NEXT
 2570 FOR K=Z TO 11:
        INPUT#2,G$:
        A$(K)="":
        FOR J=U TO LEN(G$)
 2580     A$(K)=A$(K)+CHR$(ASC(MID$(G$,J,U))-64):
      NEXT J,K:
      A$(8)="":
      A$(10)=""
 2590 FOR K=Z TO 11:
        INPUT#2,OM%(K):
      NEXT :
      FOR K=Z TO 8:
        INPUT#2,O$(K):
      NEXT
 2600 FOR K=U TO 64:
        INPUT#2,N:
        VT$=VT$+CHR$(N):
      NEXT :
      CLOSE 2
 2610 PRINT "BYTES FREE="FRE(Z)-250
 2620 INPUT "NUMBER OF LABELS";L9:
      DIM L$(L9):
      GOSUB 2700:
      RETURN

 2700 INPUT "FILENAME/'END'";G$:
      PRINT :
      IF G$="END"
      THEN RETURN

 2710 OPEN 2,1,0,G$:
      EF=Z:
      RETURN
```